UNIVERSITÀ
POLITECNICA
DELLE MARCHE

Relazione

Studenti:
Abbruzzetti Matteo
1107326 - Pallini Daniele
1100916 - Pimpini Filippo
1098388 - Rupoli Enrico
1101211 - Tridenti Noemi

Facoltà di Ingegneria - DII
Software Cybersecurity

Anno Accademico 2021 - 2022

Indice

1 Attack Resistance Analysis

1.1 STRIDE esteso

Per identificare le eventuali minacce che possono attaccare il nostro sistema, abbiamo utilizzato il modello STRIDE. Infatti, per ogni asset, abbiamo valutato quali caratteristiche di sicurezza possano essere violate e quali attacchi causino la loro violazione. STRIDE divide le varie minacce classificandole in sei tipologie:

- Spoofing (violazione dell'Authenticity);
- Tampering (violazione dell'Integrity);
- Repudiation (violazione della Non-repudiation);
- Information Disclosure (violazione della Confidentiality);
- Denial of Service (violazione dell'Availability);
- Elevation of Privilege (violazione dell'Authorization).

Nella tabella STRIDE, per ogni possibile attacco ad ogni asset, si analizza quali caratteristiche di sicurezza violano, la probabilità che hanno di accadere, l'impatto che hanno un loro eventuale accadimento, le contromisure da adottare e il loro costo, la fattibilità della realizzazione della contromisura e la loro probabilità ed impatto in seguito ad una loro eventuale realizzazione. La tabella STRIDE completa può essere visualizzata qui: tabella DUAL-STRIDE.

1.2 CAPEC & ATT&CK

Al fine di effettuare un'analisi completa di tutte le possibili minacce, abbiamo consultato CAPEC & ATTCK. CAPEC (Common Attack Pattern Enumeration and Classification) è una lista nella quale vengono classificati gli attacchi conosciuti con lo scopo di renderli noti e di migliorare le difese. ATT&CK (Adversarial Tactics, Techniques Common Knowledge) è un elenco strutturato di comportamenti noti da parte di utenti malintenzionati che sono stati compilati come tattiche e tecniche ed è utile per avere una rappresentazione dei meccanismi di attacco e di difesa.

1.3 Diagrammi i*

Il modello i* (Figura 1) permette di indicare gli attori principali, le risorse che devono essere usate o ottenute da un attore, i task che devono svolgere e gli obiettivi che vogliono raggiungere (goals e softgoals). Nel nostro sistema abbiamo individuato due attori principali, il software e l'utente. Quest'ultimo, inoltre, può essere classificato come Fornitore, Produttore o Consumatore. L'utente, attraverso il software, può richiedere l'ID utente, inserire i dati per il calcolo del carbon footprint (ad esempio emissioni dovute ad attività svolte), inserire il carbon footprint delle materie prime e ricevere il carbon footprint finale. I dati e i carbon footprint richiesti dall'utente vengono recuperati dai vari registri presenti all'interno del software, mentre la richiesta dell'ID utente viene soddisfatta attraverso un aggiornamento del log di sistema. Tutte le attività e risorse interne al software hanno come obiettivo la gestione del carbon footprint. Al fine di raggiungere questo goal si effettuano diverse attività, quali l'aggiornamento del log di sistema, la memorizzazione e restituzione del carbon footprint finale e il passaggio di proprietà dei certificati delle materie prime e dei prodotti finali. Inoltre, una corretta gestione del carbon footprint si raggiunge attraverso un altro goal, la fornitura continua del servizio. Per calcolare il carbon footprint, il sistema deve recuperare i dati dei Fornitori e i carbon footprint delle materie prime dai loro registri. L'analisi dei singoli Use case è disponibile al seguente link: Jacobson use case.

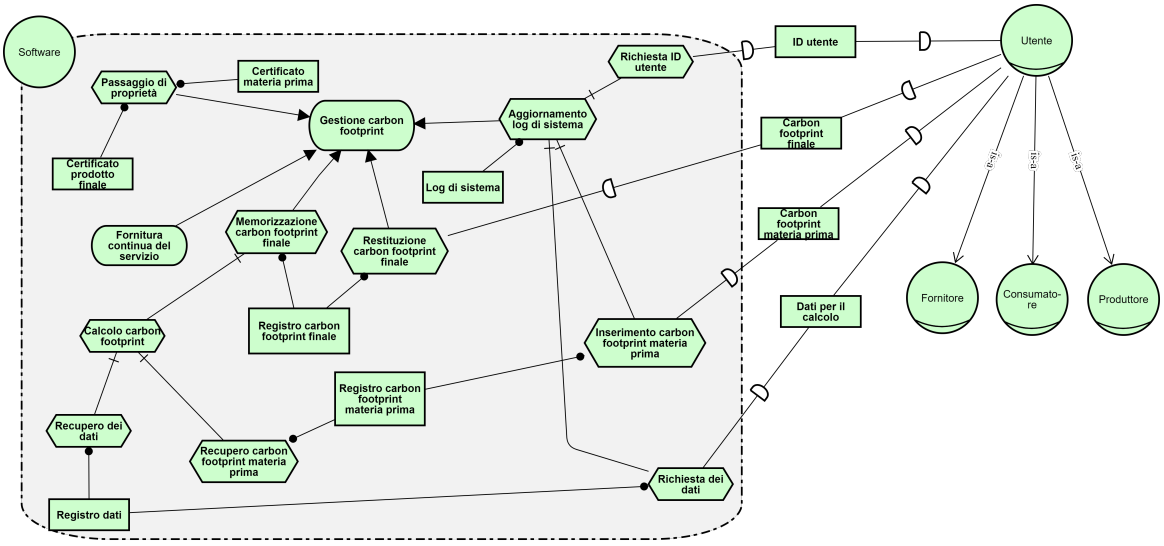

Figura 1: Diagramma I star

1.4 Attack tree

L'attack tree (Figura 2) descrive le possibili minacce che gravano sul nostro sistema e i possibili attacchi che possono portare al loro accadimento. È stato costruito un albero di attacco degli insider/outsider attacker; alla base ci sono i possibili attacchi che un utente malintenzionato può eseguire, mentre la cima dell'albero indica gli effetti che portano i vari attacchi, passando per diversi stadi di attacco.

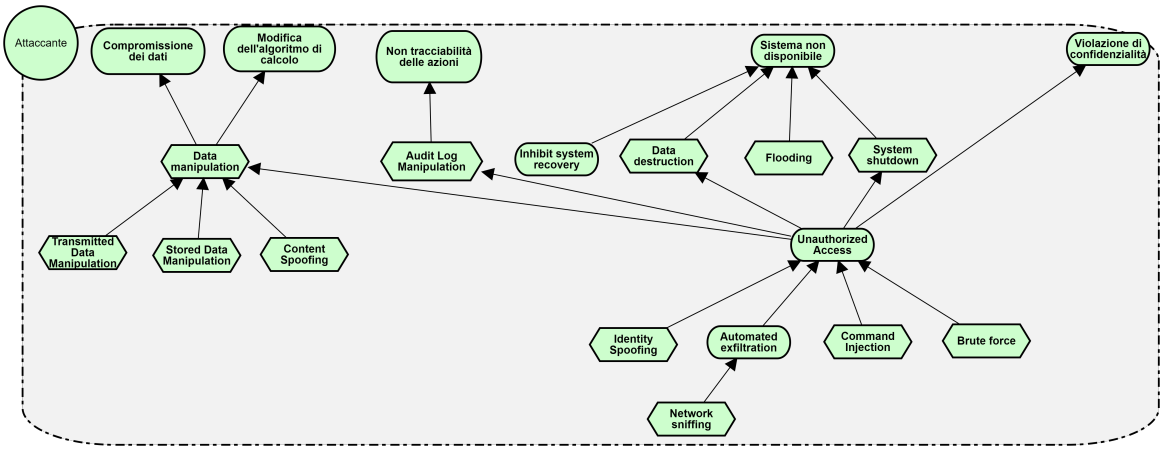

Figura 2: Attack tree

I vari attacchi vengono chiamati abuse case ed individuano le opportunità che un utente malintenzionato ha per impedire il corretto funzionamento del sistema. Gli attacchi presenti all'interno dell'attack tree vengono analizzati e valutati all'interno del seguente file: Jacobson abuse case.

1.4.1 Attack tree misuse case

L'attack tree dei misuse case identifica le varie operazioni che un utente maldestro può compiere e che può esporre il sistema in situazioni pericolose dovute alla violazione di una o più politiche di sicurezza.

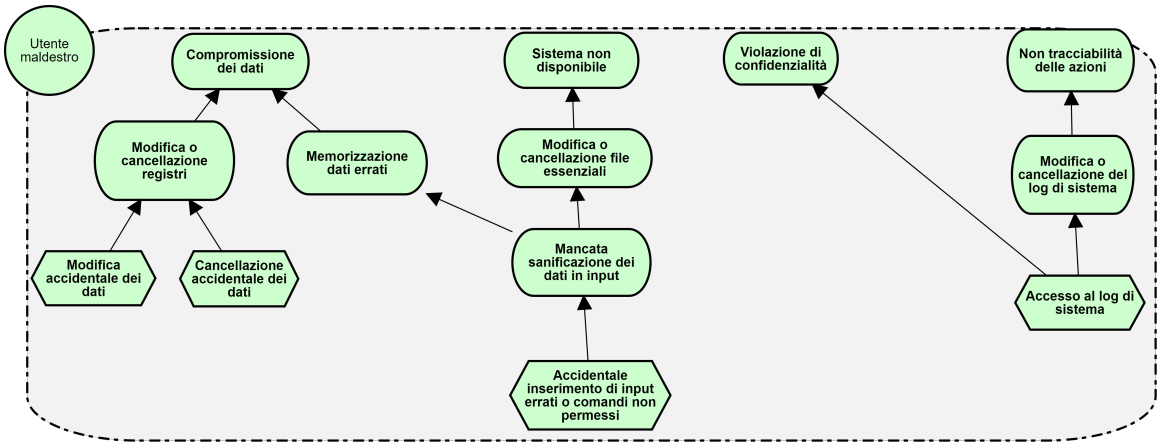

Figura 3: Attack tree misuse case

Nei misuse case è quindi presente la figura dell'utente maldestro che, attraverso un'attività mal eseguita o l'esecuzione involontaria di alcuni comandi, può causare problemi all'interno del sistema. L'elenco dei vari casi di misuse presenti all'interno dell'attack tree è stato inserito ed analizzato all'interno del seguente file: Jacobson Misuse case.

2 Design

Dopo aver eseguito la prima parte di analisi preliminare del rischio, è necessario definire il design del sistema. Sulla base di ciò che è stato analizzato e in base ai prerequisiti, si crea una prima bozza del design del sistema (architettura, scelte tecnologiche e design assets), per poi farne un'analisi del rischio e così apportarne successive modifiche e migliorarla.

2.1 Architettura

Nel nostro progetto, la tipologia di architettura che abbiamo sfruttato è quella di un sistema distribuito, ovvero senza un nodo centrale. Questo perché le reti distribuite sono le migliori dal punto di vista di tolleranza ai guasti (fault tollerance), sono sicure, trasparenti ed estremamente scalabili. Tuttavia, una rete del genere richiederà un maggiore costo di manutenzione e di sviluppo. Nello specifico, si è utilizzata una blockchain come tecnologia per registrare le transizioni peer-to-peer in una rete, nella quale i nodi assumono il ruolo di validatori. La scelta di questa tecnologia è legata alle proprietà della blockchain, nella quale le transazioni sono autentiche, immutabili, tracciabili e trasparenti, e l'utente può svolgere esclusivamente azioni a lui permesse. L'algoritmo del consenso scelto è il `raft`; esso fa parte della categoria "crash fault tollerant" e permette perciò al sistema di essere resiliente nei confronti di eventuali guasti. Inoltre, come visto nel paragrafo precedente, esistono diversi casi di abuso di un sistema che possono essere risolti mediante l'utilizzo della blockchain:

- Manipolazione del registro degli accessi: la blockchain tiene traccia di tutte le azioni effettuate in modo immutabile;
- Content spoofing del carbon footprint: con l'utilizzo di una blockchain, soltanto il software può generare il carbon footprint;
- Distruzione dei dati: utilizzando una blockchain tutti i nodi hanno una copia di tutti i dati, avendo così una ridondanza per garantirne l'integrità;

Quando si parla di design dell'architettura, uno dei problemi che si affronta è quello della protezione degli asset. Nello specifico, esistono tre tecniche difensive per mantenere la rete sicura: monitoraggio, offuscamento e isolamento.

2.1.1 Monitoraggio

Un monitor controlla e blocca l'esecuzione di un'operazione se non vengono rispettate certe policy di sicurezza. Nel nostro caso, il lavoro di monitoraggio è compiuto dalla blockchain, che si incarica di rendere le transazioni valide (controllando che l'utente abbia i privilegi necessari), immutabili, trasparenti e tracciabili. Infatti la blockchain funge da registro delle azioni, tenendo traccia di chi le compie senza possibilità di ripudio.

2.1.2 Offuscamento

Il codice e i dati sono trasmessi, e conservati, in una forma comprensibile solo a conoscenza di un segreto. Una delle tecniche utilizzate per nascondere i dati è l'hashing; all'interno del programma, si è criptato l'indirizzo dello smart contract attraverso un hash, in modo da controllarne l'integrità. In particolare, ad ogni avvio del sistema, viene effettuato un confronto tra l'hash precedente salvato e quello dell'indirizzo attualmente in memoria. Nel caso in cui il confronto dei due hash dia un esito positivo, allora è possibile utilizzare lo stesso indirizzo dello smart contract, recuperando i dati precedentemente inseriti. Qualora, invece, ci dovessero essere delle incongruenze, il sistema non è più considerato sicuro

(perché probabilmente manomesso), perciò avviene la deploy di un nuovo contratto e ne viene salvato l'indirizzo.

2.1.3 Isolamento

L'isolamento di un sistema è garantito attraverso l'utilizzo del minor numero di interfacce per accedere ad esso ed attraverso la compartimentazione degli asset. Nel nostro caso, questa tecnica è stata utilizzata isolando i vari asset del sistema (registri materie prime, prodotti, attività, carbon footprint), in modo da non rischiare che gli attori assumano privilegi non dovuti, una volta che abbiano avuto accesso ad un certo registro.

2.2 Design assets e linee guida

Un altro punto fondamentale nel design di un sistema sicuro è quello di utilizzare delle linee guida (OWASP, Sommerville, Saltzer and Schroeder) per rendere consapevole il team di ingegneri del software, quando vengono prese le decisioni di progettazione, riguardo vari problemi di sicurezza; oppure da utilizzare come checklist di revisione quando avviene la convalida del sistema. Nel nostro caso abbiamo preso in considerazione vari punti delle tre linee guide sopra citate.

2.2.1 OWASP

- PRINCIPLE OF LEAST PRIVILEGE: bisogna garantire agli utenti il minimo dei privilegi necessari e ridurre al minimo le interazioni per evitare possibili danni. Nel nostro caso, a Fornitore, Produttore e Consumatore vengono assegnati privilegi differenti e specifici al ruolo dell'utente;
- FAIL SECURELY: fallire in maniera sicura, s'intende che tutti gli errori che possono avvenire sono previsti e controllati dal software stesso. Ad esempio, nel nostro programma, le transazioni vengono controllate, ogni azione considerabile non corretta porta ad un avviso di errore e riconduce ad una successiva azione, senza bloccare il programma;
- SEPARATION OF DUTIES: si ricollega al principio dei minimi privilegi; ovvero come ogni ruolo permette di avere differenti privilegi, questo porterà ad avere anche differenti doveri. Nel nostro caso,ad esempio, non possiamo permettere al Fornitore di acquistare prodotti finiti, o al Consumatore di inserire materie prime;
- KEEP SECURITY SIMPLE: bisogna fare in modo che il sistema sia il più semplice possibile, anche perché strutture troppo complicate possono portare ad errori nell'utilizzo.

2.2.2 Sommerville

- LOG USER ACTION: mantenere un registro delle azioni dell'utente, che può essere analizzato per scoprire chi ha fatto cosa. Se gli utenti sono a conoscenza di un tale registro, è meno probabile che si comportino in modo irresponsabile. Nel nostro caso, tutte le azioni compiute da Fornitore/Produttore/Consumatore vengono salvate all'interno della blockchain;
- USE REDUNDANCY AND DIVERSITY TO REDUCE RISK: conservare più copie dei dati e utilizzare infrastrutture diverse in modo che una vulnerabilità dell'infrastruttura non possa essere l'unico punto di errore. Nel nostro caso, la ridondanza si ottiene utilizzando la blockchain, che registra ogni transazione e ogni dato salvato può essere distribuito, in copia, ai vari nodi membri della rete;

- SPECIFY THE FORMAT OF ALL INPUTS: nel nostro caso, il format di tutti gli input viene controllato e sanificato. Se si eccede in dimensione (overflow) o se vengono inseriti caratteri diversi da quelli previsti, il sistema fa una richiesta di reinserimento dei dati. Ad esempio, nella parte off-chain abbiamo utilizzato la libreria JavaScript "Validator" per controllare la tipologia di input: per l'inserimento dei nomi delle materie prime, dei prodotti, delle attività o dei lotti vengono richiesti caratteri alfanumerici, mentre nel caso delle emissioni unicamente caratteri numerici. Un altro esempio, è l'utilizzo della libreria "Number" per controllare il tipo di input numerico, ovvero se il numero inserito non sia negativo, troppo grande (overflow) o non intero;
- COMPARTMENTALIZE YOUR ASSETS: organizzare tutti gli assets in aree separate, cosicché soltanto gli utenti autorizzati possano farvi uso. Nel nostro progetto, ad esempio, abbiamo isolato i vari registri relativi ai diversi asset, quali materie prime, prodotti e carbon footprint. In questo modo, soltanto gli utenti autorizzati possono accedervi e si garantisce così un maggiore livello di sicurezza.

2.2.3 Saltzer & Schroeder

- PSYCOLOGICAL ACCEPTABILITY/ EASY TO USE: l'interfaccia dovrà essere sviluppata in modo da rimanere semplice ed intuitiva per l'utente, così da evitare ulteriori errori a livello umano.

2.3 Weakness Analysis

Dopo un primo design del sistema, occorre fare un'analisi delle debolezze che sorgono facendo affidamento su software esterni. Ci viene in aiuto, per questa analisi, il catalogo CWE (Common Weakness Enumaration) del MITRE, una lista sviluppata dalla community delle più comuni debolezze a livello hardware e software. Si sono riscontrate diverse possibili debolezze del nostro sistema.

2.3.1 Improper access control

Il software non limita o limita in modo errato l'accesso ad una risorsa ad un attore non autorizzato. Il controllo degli accessi prevede l'utilizzo di diversi meccanismi di protezione quali:

- Autenticazione: dimostrare l'identità di un attore; nel nostro programma, si richiede all'utente esclusivamente di identificarsi come Fornitore, Produttore o Consumatore;
- Autorizzazione: assicurare che un determinato attore possa accedere alle risorse (materie prime, prodotti, bilanci, carbon footprint e attività) e alle funzionalità a lui destinate;
- Non-ripudio: tracciamento delle attività svolte; nel nostro caso, nella blockchain vengono registrate tutte le transazioni svolte dai vari utenti.

Quando uno di questi meccanismi non viene applicato, o fallisce, gli aggressori potrebbero compromettere la sicurezza del software, ad esempio acquisendo privilegi, leggendo informazioni sensibili, eseguendo comandi o eludendo il tracciamento.

2.3.2 Protection mechanism failure

Il protection mechanism failure si verifica quando un sistema non utilizza, o utilizza in modo non corretto, un meccanismo di protezione al fine di avere una difesa sufficiente contro gli attacchi diretti al prodotto. Questa debolezza copre situazioni distinte:

1. Un meccanismo di protezione "mancante" si verifica quando l'applicazione non definisce alcun meccanismo contro una determinata classe di attacchi. Ad esempio, quando il software non cifra le informazioni sensibili prima dell'archiviazione o della trasmissione. La mancanza di un'adeguata crittografia dei dati perde le proprietà di riservatezza, integrità e responsabilità che una crittografia correttamente implementata garantisce;

2. Un meccanismo di protezione "insufficiente" potrebbe fornire alcune difese, ad esempio contro gli attacchi più comuni, ma non protegge da tutto. Si ha una debolezza quando il software archivia o trasmette dati sensibili utilizzando uno schema di crittografia teoricamente valido, ma non abbastanza forte per il livello di protezione richiesto, infatti uno schema di crittografia debole può essere soggetto ad attacchi di forza bruta. Un'altra possibile debolezza si ha quando il software non verifica a sufficienza l'origine o l'autenticità dei dati, accettando così dati non validi. Per questo motivo, nel nostro programma abbiamo fatto in modo che ogni azione di inserimento (materie prime, prodotti, attività) e ogni trasferimento di token (acquisto materie prime, prodotti intermedi e prodotti finiti) siano tracciati dalla blockchain.

2.3.3 Improper control of a resource through its lifetime

Si verifica quando il software non mantiene, o mantiene in modo errato, il controllo su una risorsa per tutta la sua durata di creazione, utilizzo e rilascio. Le risorse hanno spesso istruzioni esplicite su come essere create, utilizzate e distrutte. Quando il software non segue queste istruzioni, può portare a comportamenti imprevisti e stati potenzialmente sfruttabili. Anche senza istruzioni esplicite, ci si aspetta che vengano rispettati vari principi, come "non utilizzare un oggetto fino al completamento della sua creazione", nel nostro caso, non si può comprare un prodotto finito prima della fine del processo di lavorazione e del calcolo del carbon footprint finale; oppure "non utilizzare un oggetto dopo che è stato programmato per la distruzione". Ad esempio, nel nostro programma, quando un token non deve essere più utilizzato viene invalidato.

2.3.4 Incorrect calculation

Si verifica quando il software esegue un calcolo che genera risultati errati o non intenzionali, che vengono successivamente utilizzati nelle decisioni critiche per la sicurezza o nella gestione delle risorse. Quando il software esegue un calcolo critico per la sicurezza in modo errato, potrebbe causare allocazioni di risorse errate, calcoli non corretti, assegnazioni di privilegi errate o confronti non riusciti. Molti dei risultati diretti di un calcolo errato possono portare a problemi ancora più grandi, come meccanismi di protezione falliti o persino esecuzione arbitraria di codice. Nel nostro caso, ad esempio, errori di calcolo possono portare ad una valutazione errata del carbon footprint finale, e quindi una inaccettabile mancanza di integrità nella valutazione dei prodotti finali.

2.3.5 Improper check or handling of exceptional conditions

Si ha quando il software non anticipa o gestisce non correttamente condizioni eccezionali, che si verificano raramente durante il normale funzionamento del software. Nel nostro caso, tutti le azioni impreviste che possono accadere sono gestite dal programma stesso, con messaggi d'errore appropriati e senza bloccare il flusso del software.

2.3.6 Improper neutralization

Si verifica quando il prodotto non garantisce, o garantisce in modo errato, che i messaggi o i dati strutturati siano ben formati e che determinate proprietà di sicurezza siano soddisfatte prima di essere

letti da un componente a monte o inviati a un componente a valle. Se un messaggio non è corretto, è possibile che il messaggio venga interpretato in modo errato. Neutralizzazione è un termine astratto per qualsiasi tecnica che garantisca che l'input,e l'output, sia conforme alle aspettative e sia "sicuro". Questa può essere:

1. verificare che l'input/output sia già "sicuro" (es. validazione);
2. trasformare l'input/output in "sicuro" utilizzando tecniche quali filtraggio e codifica/decodifica;
3. impedire che l'input/output venga fornito direttamente da un utente malintenzionato (es. "selezione indiretta" che mappa i valori forniti dall'esterno ai valori controllati internamente);
4. impedire che l'input/output venga elaborato.

Nel nostro caso, sono state applicate tecniche di filtraggio e validazione degli input, verificandone la correttezza dei formati. La procedura di sanificazione effettuata a monte della blockchain non permette che dati non conformi siano inseriti al suo interno e potenzialmente compromettano la sicurezza e l'operabilità del sistema. Essendo i dati in input sanificati, si ha la certezza che anche quelli in output rispettino i formati corretti.

3 Smart Contract

3.1 I token

Il contratto sviluppato contract.sol implementa token non fungibili utilizzando lo standard ERC-721 importato dalla libreria omonima di OpenZeppelin. Ad ogni inserimento dell'utente viene incrementato un contatore e viene minato un token, che ha come identificativo un numero derivato dal contatore; l'intero è a sua volta l'indice di un hash table, che ha come elementi le strutture contenenti il nome del prodotto inserito, il lotto, le relative emissioni di CO2 ed altre informazioni che verranno mostrate in seguito. I token vengono minati attraverso la funzione `_mint()` e sono divisi in tre categorie: le materie prime, i prodotti e i carbon footprint. La divisione è implementata attraverso l'utilizzo di un prefisso aggiunto al contatore corrispondente; tutti i token relativi ad una materia prima hanno come prefisso "1", quelli relativi ad un prodotto hanno il prefisso "2" e quelli relativi ad un carbon footprint hanno il prefisso "3". Per cui ci sono tre diversi contatori e tre diverse hash table, implementate attraverso la funzione di mapping.

```
mapping(uint=>MateriaPrima) listaMateriePrime;
mapping(uint=>Prodotto) listaProdotti;
mapping(uint=>Prodotto) listaCarbonFootprint;
```

Le materie prime sono definite da una struttura contenente il nome della materia prima, il lotto e le relative emissioni; i prodotti contengono le stesse informazioni con l'aggiunta delle emissioni di produzione e delle informazioni sulle materie prime da cui è derivato; il carbon footprint è il token finale che viene venduto al cliente, esso ha la stessa struttura di un prodotto ma, come vedremo, le sue emissioni sono comprensive di tutte le attività svolte su di esso. Al momento della trasformazione di una o più materie prime a prodotto, i token relativi a quelle materie prime vengono invalidati attraverso la funzione `_burn()`, lo stesso avviene all'interno della funzione di trasformazione di un prodotto in un carbon footprint. Oltre alle funzioni per l'inserimento di nuovi token, sono presenti delle funzioni di tipo call, attraverso le quali è possibile ricercare all'interno delle hash table. In particolare si può effettuare una ricerca per nome oppure per nome e lotto; è possibile, inoltre, visualizzare tutti i token posseduti o quelli acquistabili. In particolare, la funzione di ricerca è realizzata scorrendo una lista desiderata, ad esempio `listaMateriePrime`. Si confronta la stringa in ingresso alla funzione search con tutti i nomi delle materie prime appartenenti alla lista. Se necessario viene comparato anche il lotto. Gli indici dell'hash table per cui si verifica una corrispondenza vengono inseriti all'interno di un array, i cui elementi vengono poi utilizzati per ricavare il token corrispondente, verificarne l'esistenza ed il proprietario. Se i dati coincidono a quelli di ricerca la funzione restituisce i relativi token.

3.2 Gli utenti

Attraverso l'utilizzo di un costruttore si forniscono al contratto le variabili di stato costituite dagli indirizzi del Fornitore, del Produttore e del Consumatore. In questo modo è possibile definire quali utenti possono eseguire certe azioni. In particolare, attraverso l'utilizzo dei `require`, si può determinare se una certa funzione può essere richiamata da un utente o meno. Per esempio, la funzione `inserimentoMateriaPrima()` può essere eseguita solamente dal Fornitore per cui al suo interno troviamo un comando del tipo: `require(msg.sender==Fornitore, ''Solo il Fornitore può inserire una materia prima'')`. Attraverso questo meccanismo ogni utente ha il minimo dei privilegi possibili.

3.3 La gestione delle attività

Il Produttore, che è il proprietario dello smart contract, può svolgere delle attività su una materia prima, come ad esempio il trasporto, oppure direttamente sul prodotto. Per questo motivo si è definita un nuovo mapping contenente i cosiddetti task.

```
mapping(uint=>Task[]) listaTask;
```

In questo caso, l'indice dell'hash table corrisponde al numero del token a cui fa riferimento. Ogni elemento è un array di task, in quanto è possibile effettuare più attività sulla stessa materia prima o sullo stesso prodotto. Una volta che il Produttore ha concluso ogni attività, ed il prodotto è pronto per la vendita, è possibile calcolarne il carbon footprint, che è il token finale comprensivo di tutte le emissioni.

3.4 Analisi statica

Per eseguire l'analisi statica del contratto si è utilizzato il plugin di Remix "SOLIDITY STATIC ANALYSIS", il quale permette di controllare la presenza di vulnerabilità o di errate pratiche di sviluppo. Gli avvisi riportati dal tool sono per la maggior parte riconducibili allo standard ERC-721, per i restanti si riportano i seguenti warning:

- `Gas costs: Gas requirement of function <function> is infinite.` Come per le transazioni presenti in ERC-721, Remix non è in grado di prevederne il gas necessario, perciò ne associa gas infinito;
- `Loop over dynamic array: Loops that do not have a fixed number of iterations.` Al momento dell'inserimento di alcuni input utente, si effettua un ciclo for per elaborare ogni ingresso inserito. Il problema non sussiste in quanto l'utente può inserire solamente un numero finito di input;
- `Constant/View/Pure functions: <function>: Is constant but potentially should not be.` L'avviso si presenta ogni volta che si definisce una funzione di tipo view. È stato manualmente verificato che all'interno di tali funzioni non si modificano variabili globali;
- `Similar variable names: Variables have very similar names.` All'interno del codice sono state utilizzate variabili simili appositamente;
- `Guard conditions: Use "assert(x)" if you never ever want x to be false.` L'avviso si presenta ogni volta che si utilizza una `require`; la scelta deriva dalla necessità di verificare condizioni a runtime e non in fase di progettazione.

3.5 Collegamento alla blockchain

Per la gestione dello smart contract si è scelto di utilizzare "Web3.js", una collezione di librerie che permette di interagire con i nodi della blockchain Quorum, utilizzando il linguaggio JavaScript. La compilazione del codice Solidity viene effettuata sull'Ethereum IDE Remix, dalla quale è possibile salvare l'ABI e il bytecode del contratto. Salvando queste due variabili all'interno del codice JavaScript, è possibile richiamare le funzioni dello smart contract in semplici passaggi. Le interazioni con il contratto sono possibili attraverso le call, che permettono di visualizzare informazioni salvate sulla blockchain senza modificarne lo stato, ed attraverso le transazioni, che generano invece un nuovo blocco, ad esempio minando un nuovo token o trasferendolo da un utente all'altro. Gli utenti che accedono alla blockchain sono tre: Fornitore, Produttore e Consumatore. Si è scelto perciò di utilizzare una blockchain con tre

nodi, ognuno dei quali ha un solo account con il quale autenticarsi. Il collegamento verso i nodi avviene attraverso l'accesso a delle specifiche porte raggiungibili tramite `localhost`, nello specifico le porte sono la 22000, la 22001 e la 22002.

```
let web3fornitore = new Web3('http://localhost:22000');
let web3produttore = new Web3('http://localhost:22001');
let web3consumatore = new Web3('http://localhost:22002');
```

Nella pratica questo vuol dire che Fornitore, Produttore e Consumatore effettuano il login in tre diversi terminali, mentre, utilizzando nodi virtuali appartenenti alla stessa macchina, l'autenticazione con un certo account equivale al connettersi al relativo nodo. Il manuale contenente tutte le istruzioni per l'installazione e l'utilizzo del software è disponibile nella Guida all'uso.

3.6 Acquisto di materie prime e prodotti

Due delle transazioni possibili sono inerenti all'acquisto, da parte del Produttore e del Consumatore, rispettivamente di materie prime e prodotti. Il trasferimento di un token da parte dello smart contract, deve essere invocato dal proprietario, quindi da colui che vende la materia prima o il prodotto. Nel codice sviluppato, invece, si dà la possibilità all'acquirente di effettuare un'azione di acquisto. Ciò è più pratico per un'implementazione reale.

3.7 Il problema dell'overflow

Un possibile problema nel calcolo delle emissioni è l'overflow. Esso può avvenire in due modi: l'utente inserisce un valore di emissioni non computabile dallo smart contract, oppure è la somma di più campi emissioni ad essere superiore a $2^{256} - 1$, ovvero il valore massimo assumibile da un intero nello smart contract. Il primo caso viene gestito al momento dell'inserimento dei dati: viene utilizzata la funzione `isSafeInteger()` della libreria "Number" di Node.js, la quale controlla che il numero inserito sia minore di $2^{53} - 1$. Per prevenire il secondo caso, viene utilizzata invece una libreria di OpenZeppelin chiamata "SafeMath", al cui interno è disponibile una funzione `add()`:

```
function add (uint256 a, uint256 b) internal pure returns (uint256) {
    uint256 c = a + b;
    require (c >= a, "SafeMath: addition overflow");

    return c;
}
```

Ogni volta che è necessario sommare emissioni, come per esempio durante l'inserimento di un prodotto, si controlla che la somma sia superiore di uno dei due addendi. In questo modo è possibile prevenire un overflow. Per quanto riguarda i contatori, la libreria Counters di OpenZeppelin implementa nativamente la stessa funzione.